

# **CS 401**

## **Divide and Conquer**

Xiaorui Sun

# Stuff

Midterm grade will be released this afternoon


A course questionnaire sent to you this morning

- Let me know if you have any suggestions/concerns

# Master Theorem

# Master Theorem

Suppose  $T(n) = a T\left(\frac{n}{b}\right) + cn^k$  for all  $n > b$ . Then,



$c$ : absolute constant

- If  $a < b^k$  then  $T(n) = \Theta(n^k)$
- If  $a = b^k$  then  $T(n) = \Theta(n^k \log n)$
- If  $a > b^k$  then  $T(n) = \Theta(n^{\log_b a})$

Works even if it is  $\left\lceil \frac{n}{b} \right\rceil$  instead of  $\frac{n}{b}$ .

We also need  $a \geq 1, b > 1, k \geq 0$  and  $T(n) = O(1)$  for  $n \leq b$ .

# Question

Consider the following recurrence. Which case of the master theorem applies?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 3T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- A.  $T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585})$
- B.  $T(n) = \Theta(n \log n)$
- C.  $T(n) = \Theta(n)$
- D. Master theorem not applicable

## Master Theorem

Suppose  $T(n) = a T\left(\frac{n}{b}\right) + cn^k$  for all  $n > b$ .

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

# Question

Consider the following recurrence. Which case of the master theorem?

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(n - 3\left\lfloor \frac{n}{10} \right\rfloor\right) + \frac{11}{5}n & \text{if } n > 1 \end{cases}$$

Master Theorem

Suppose  $T(n) = a T\left(\frac{n}{b}\right) + cn^k$  for all  $n > b$ .

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

- A.  $T(n) = \Theta(n)$
- B.  $T(n) = \Theta(n \log n)$
- C.  $T(n) = \Theta(n^2)$
- D. Master theorem not applicable

Akra–Bazzi theorem  
Wiki!

# How to use master theorem?

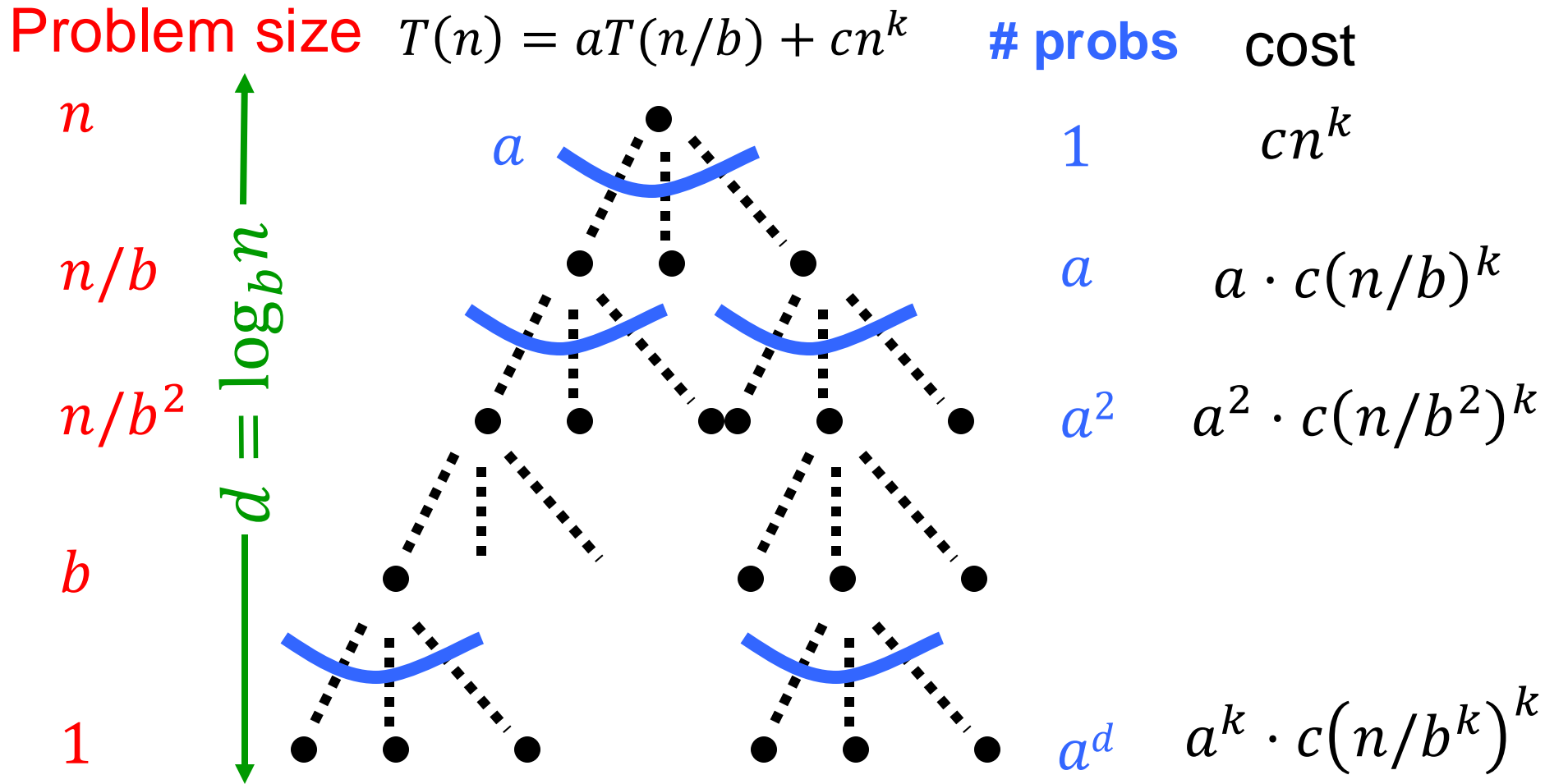
For a divide and conquer algorithm

- $a$ : number of subproblems
- $b$ : ratio of problem size / subproblem size
- $c \cdot n^k$ : running time of divide and combine step

We have recurrence  $T(n) = a T\left(\frac{n}{b}\right) + cn^k$  for all  $n > b$ .

Example: Mergesort have two subproblems of half size of the original problem, and the cost of divide and combine step is  $O(n)$ , so  $T(n) = 2 T(n / 2) + c n$ , which implies  $T(n) = \Theta(n \log n)$

# Understand Master Theorem



$$T(n) = \sum_{i=0}^{d=\log_b n} a^i c \left( \frac{n}{b^i} \right)^k$$



# Understand Master Theorem

Suppose  $T(n) = a T\left(\frac{n}{b}\right) + cn^k$  for all  $n > b$ . Then,

- If  $a < b^k$  then  $T(n) = \Theta(n^k)$  # of problems increases **slower** than the decreases of cost.  
**Top** term dominates.
- If  $a = b^k$  then  $T(n) = \Theta(n^k \log n)$
- If  $a > b^k$  then  $T(n) = \Theta(n^{\log_b a})$  # of problems increases **faster** than the decreases of cost  
**Bottom** term dominates.

$$T(n) = \sum_{i=0}^{d=\log_b n} a^i c \left(\frac{n}{b^i}\right)^k$$

# Binary Search

# Search Algorithms

Search: Given an element and an array, is the element in the array?

53	72	14	97	33	93	51	6	96	10	84	45	95	64	25
----	----	----	----	----	----	----	---	----	----	----	----	----	----	----

Search for 96: Yes

Search for 11: No

Naïve algorithm: Sequential search

- $O(n)$  running time

Can we do better?

# Let us try divide and conquer

## Divide and conquer

- Divide: separate list into two pieces.
- Conquer: recursively find the required element in each half.
- Combine: return yes if any subproblem returns yes, otherwise, no

53	72	14	97	33	93	51	6	96	10	84	45	95	64	25
----	----	----	----	----	----	----	---	----	----	----	----	----	----	----

Divide:  $O(1)$ .

53	72	14	97	33	93	51	6
----	----	----	----	----	----	----	---

96	10	84	45	95	64	25
----	----	----	----	----	----	----

Conquer:  $2T(n / 2)$

Search for 96: No

Yes  $\Rightarrow$

Yes

Search for 11: No

No  $\Rightarrow$

No

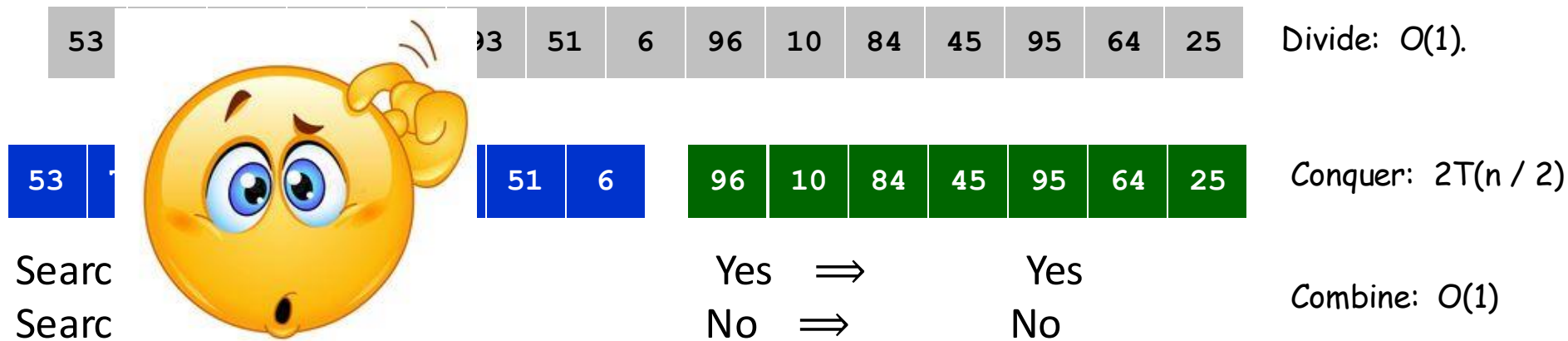
Combine:  $O(1)$

$$T(n) = 2T(n/2) + O(1)$$

# Let us try divide and conquer

## Divide and conquer

- Divide: separate list into two pieces.
- Conquer: recursively find the required element in each half.
- Combine: return yes if any subproblem returns yes, otherwise, no



Not better than sequential search

Overall:  $O(n)$

If no additional condition,  $O(n)$  is the best to hope

# Let us try divide and conquer

## Assume the array is sorted

- Divide: separate list into two pieces.
- Conquer: **Key point: avoid recursively solve both**
- Combine: return yes if any subproblem returns yes, otherwise, no

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Divide:  $O(1)$ .

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Conquer:  $T(n / 2)$

Search for 96: No need to recurse on the first half

- Largest number in the first half  $< 96$

Search for 11: No need to recurse on the second half

- Smallest number in the second half  $> 11$

Combine:  $O(1)$

**$O(1)$  time to decide which subproblem to solve!**

# Binary Search

Invariant. Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

BinarySearch(*A*, *low*, *high*, *key*)

```
if high < low:  
    return No  
mid  $\leftarrow \left\lfloor \text{low} + \frac{\text{high} - \text{low}}{2} \right\rfloor$   
if key = A[mid]:  
    return mid  
else if key < A[mid]:  
    return BinarySearch(A, low, mid - 1, key)  
else:  
    return BinarySearch(A, mid + 1, high, key)
```

# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ <b>low</b>														↑ <b>high</b>

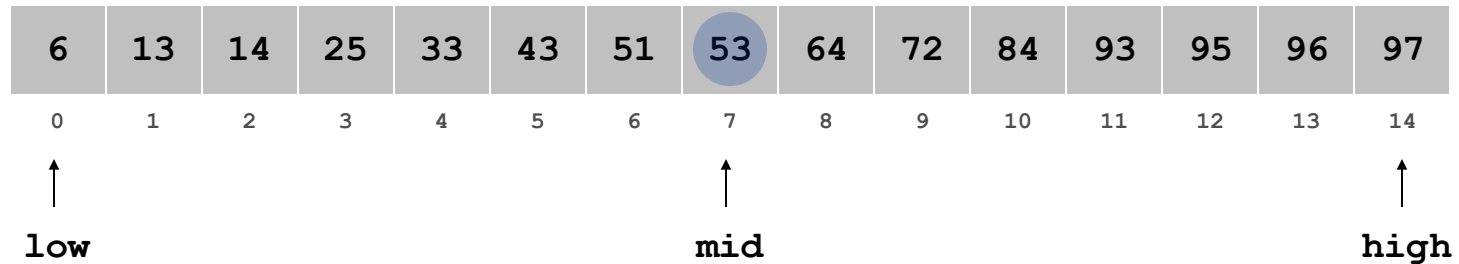


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.

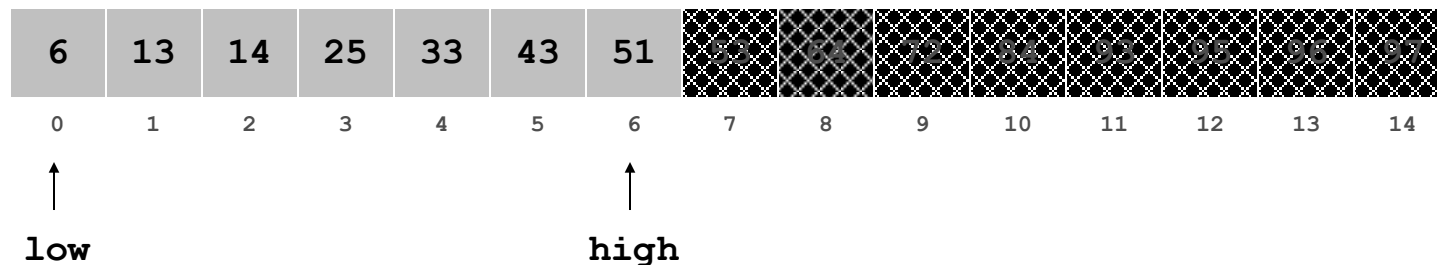


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

Ex. Binary search for 33.

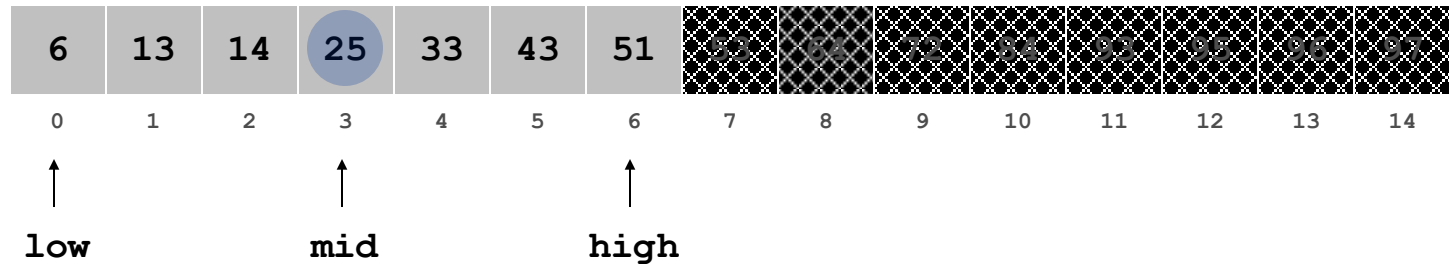


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.

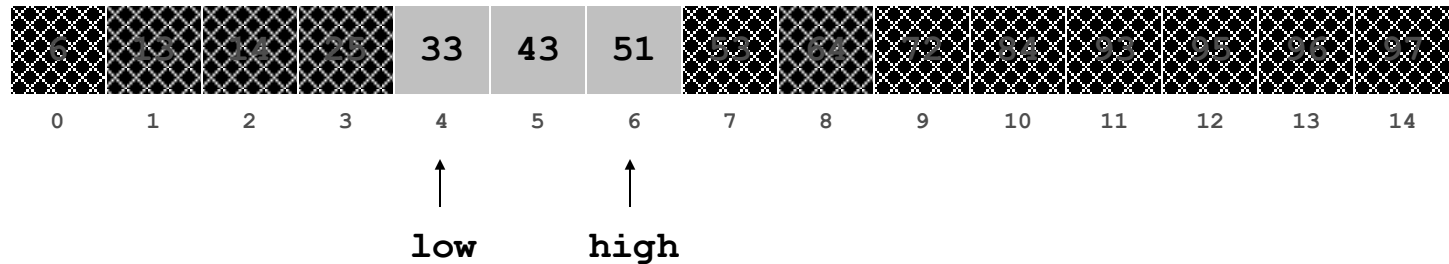


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.

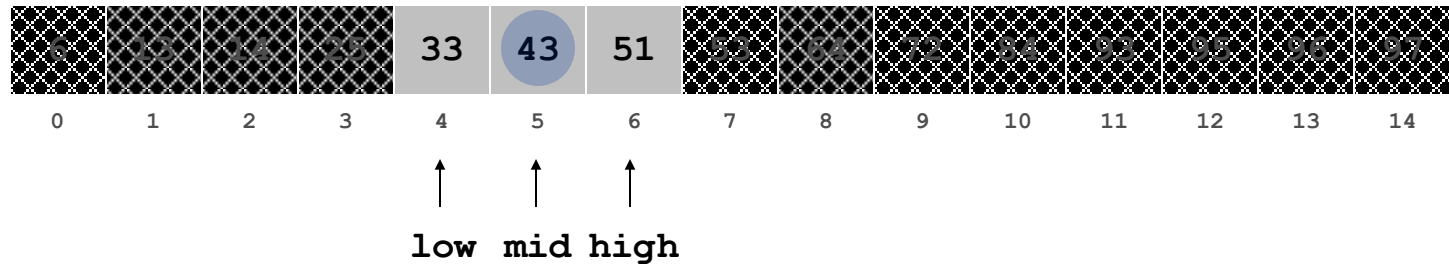


# Binary Search

**Binary search.** Given  $key$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = key$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[low] \leq key \leq A[high]$ .

**Ex.** Binary search for 33.

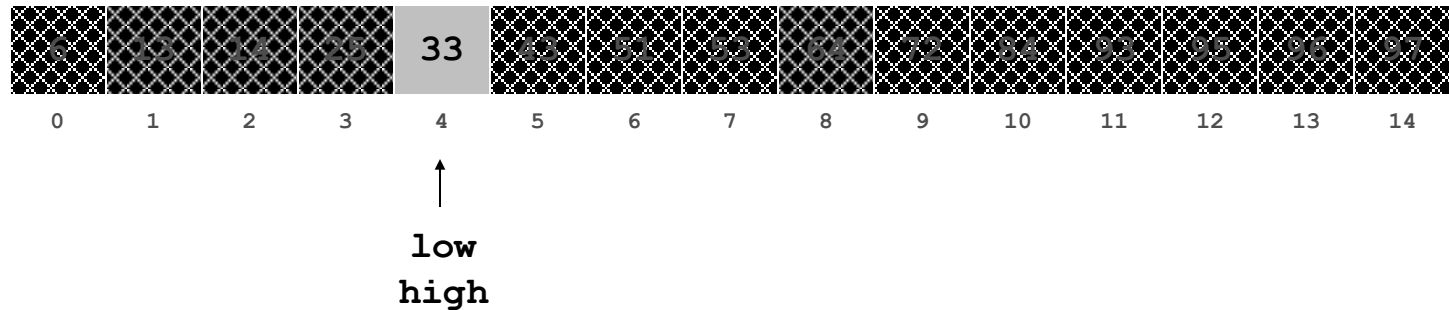


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.

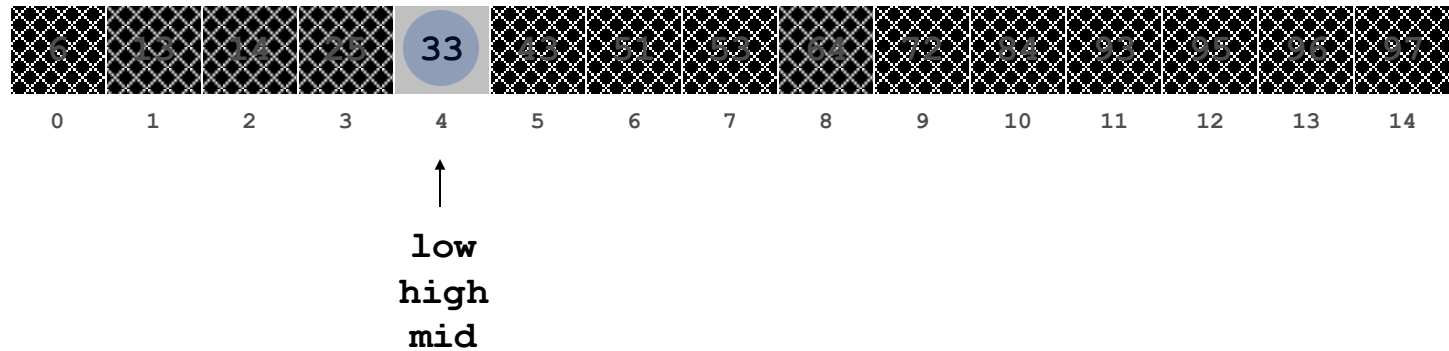


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.

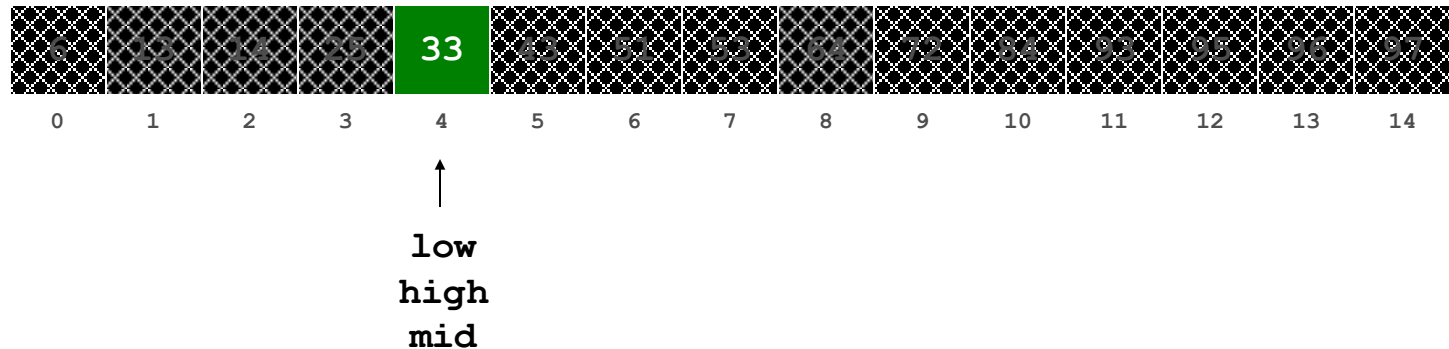


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 33.



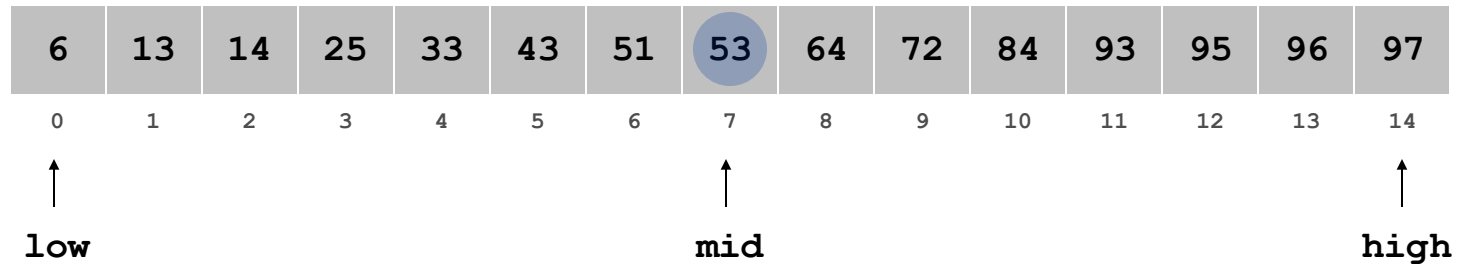


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 47.

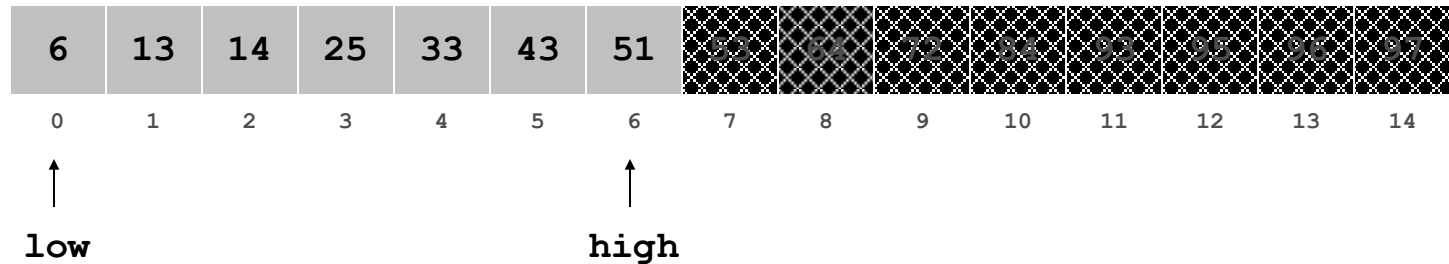


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

Ex. Binary search for 47.

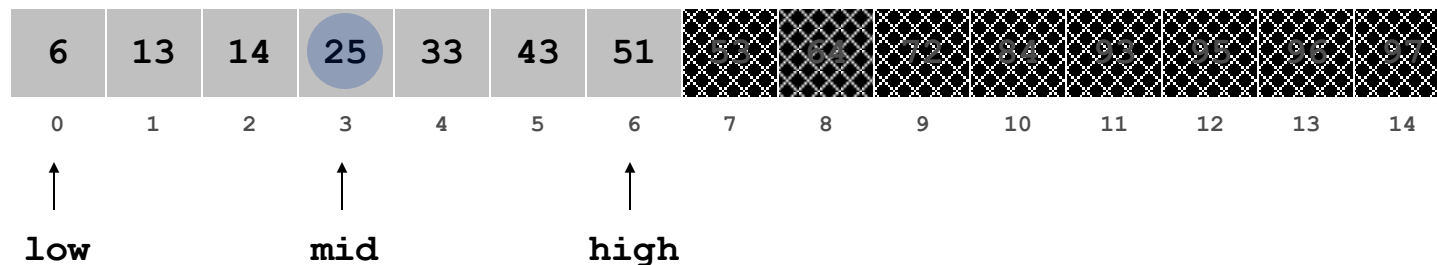


# Binary Search

**Binary search.** Given `key` and sorted array `A[]`, find index `i` such that `A[i] = key`, or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

Ex. Binary search for 47.

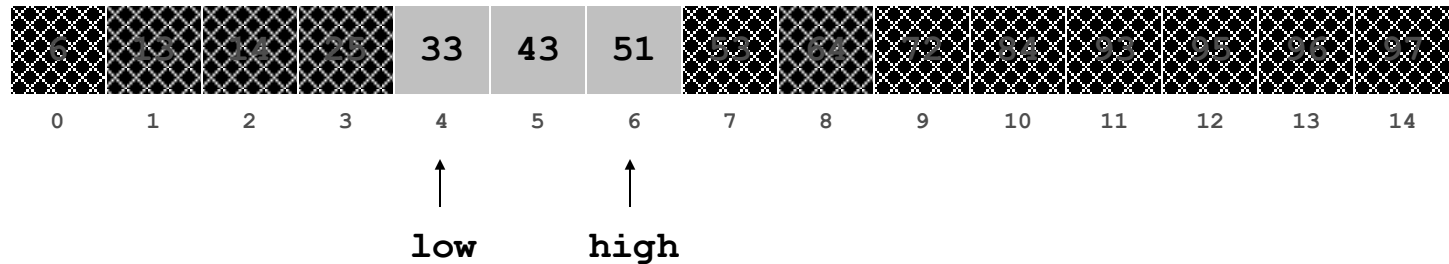


# Binary Search

**Binary search.** Given  $key$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = key$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[low] \leq key \leq A[high]$ .

**Ex.** Binary search for 47.

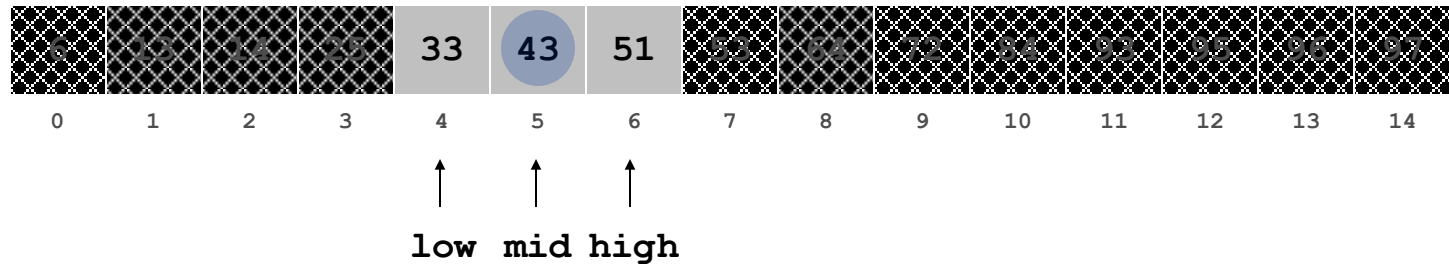


# Binary Search

**Binary search.** Given  $key$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = key$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[low] \leq key \leq A[high]$ .

**Ex.** Binary search for 47.

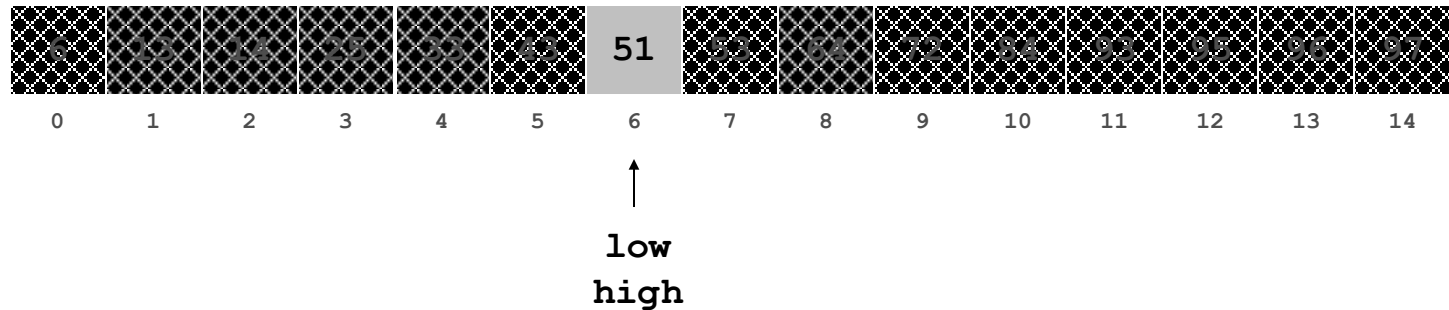


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 47.

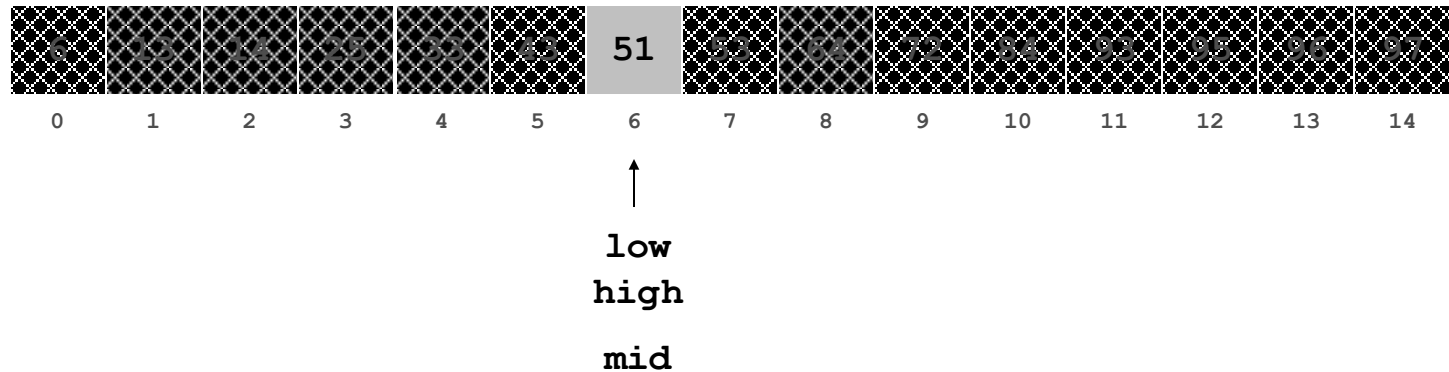


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 47.

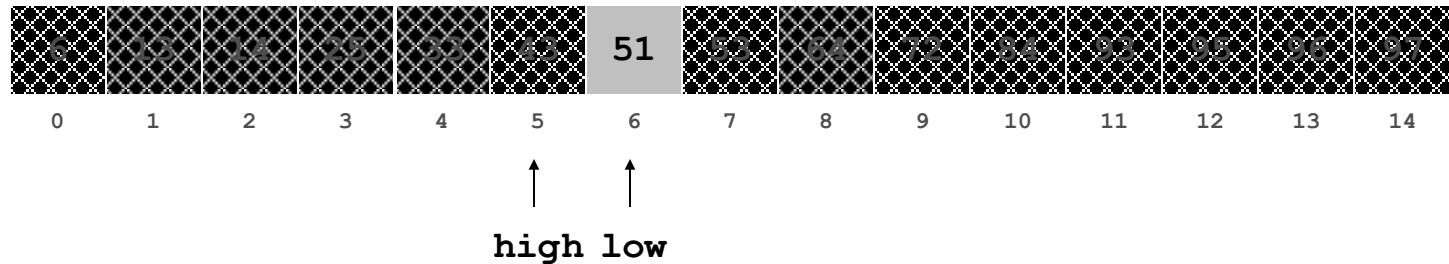


# Binary Search

**Binary search.** Given  $\text{key}$  and sorted array  $A[]$ , find index  $i$  such that  $A[i] = \text{key}$ , or report that no such index exists.

**Invariant.** Algorithm maintains  $A[\text{low}] \leq \text{key} \leq A[\text{high}]$ .

**Ex.** Binary search for 47.



47 is not in the array



# Binary Search

Binary search running time

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$
$$\Rightarrow T(n) = O(\log n)$$

Lesson: Additional structure (sorted array) can break the usual lower bound